V.04.

Общий обзор занятия.

Начинаем строить структуры данных “вручную”. И обязательно сразу проводим линию:

```
задача -> метод (алгоритм) решения -> необходимые способы доступа к данным -> ко
нструируем соответствующую реализацию структуры данных
```

Нас интересует прежде всего не просто сконструировать структуру данных, а забросить детей в связные структуры. Тут проблемка есть: линейные структуры как правило вполне себе эффективно реализуются на слайсах, и это детям проще и понятнее. Однако давать сразу какие-то разветвлённые структуры - совсем не вариант. Так что в этот раз

- Рассмотрим пару задач
- Попридумываем для них решения,
- а затем будем искать способы представления/хранения данных, по возможности максимально адекватные тем способам доступа к данным, которые требует придуманный способ решения:
    - набор необходимых способов - это, конечно, и есть абстрактный тип данных, интерфейс структуры; всё это так, но пока я бы избегал такого “богатого” слова - интерфейс, понимания оно не добавляет, так что ни к чему в данном случае множить сущности
    - а дальше конструируем реализацию придуманного абстрактного типа данных, стараясь подобрать максимально адекватные техники хранения.

Сегодня построим стек (тщательно) и очередь(по-быстрому). Конечно, обойти их реализацию на слайсах, которые фактически есть плюсовые вектора, только лучше, и неправильно, и просто невозможно. Но, кроме реализации этих АТД на слайсах, реализуем их на списке, на односвязном списке.

Насколько сильно подчёркивать отделение набора операций, набора способов доступа к данным от их реализации решается по ходу пьесы - по обстоятельствам, как занятие пойдёт, но при любых обстоятельствах этот мотив должен прозвучать

очень ясно, явно и внятно.

В итоге получается в занятии довольно много принципиального момента, да. Но всё-таки технический момент в этот раз очень важен, даже, пожалуй, он важнее принципиального. А технический момент - это работа с указателями, причём не просто с адресами, как это было на предыдущих занятиях, а именно с указателями, со структурообразующими элементами хранения. И тут, конечно, детям будет больно для начала, но обойти это место нельзя, это они должны понять, осознать и постараться преодолеть эти трудности поскорее. А, поскольку трудности чисто технические, то имеется единственный способ их преодолевать - упражнения.

Оглавление. Основные вопросы

- Задача о проверке корректности скобочного выражения
    - Построение алгоритма решения. На этом фоне получаем, что необходимый для реализации полученного алгоритма набор операций определяет стек (LIF0). Коротко об абстрактных типах данных (АТД) - полностью определяется необходимым набором способов доступа к элементам струкутуры, фактически АТД - это и есть набор способов доступа.
    - Реализуем стек на слайсе. Здесь вспоминаем про методы, про из сообенности. Пишем методы типа stack, пишем решение основной задачи.
    - Но слайс - явно избыточная струкутура данных. Она даёт возможность непосредственно обращаться к любому элементу по его номеру, вставлять или удалять элементы в середине списка, а нам это лишнее - нам достаточно иметь доступы только на конце списка. Приходим к идее связного хранения данных.
    - Реализуем стек на связном списке. Опять методы.
    - Особенность! Указательные типы не могут иметь методов. Так что прячем указательный тип в struct с одним полем.
- Задача о максимуме в плавающем окне
    - Решенеие. Очевидная очередь.
    - Строим способы доступа, реализуем очередь на слайсе.
    - Выясняем, что нам опять хватает односвязного списка для реализации очереди. Только надо бы держать этот список за две точки - за “голову” и за “хвост”.
    - Реализуем очередь на связном списке. Опять оформляем реализацию как набор методов.

- Практика
    - Задачи на построение “хитросвязанных” связных конструкций
    - Задачи на реализацию методов типа linkedList

Лекция

Задача о скобках - стек.

Описание задачи

Начнём с задачи о скобках одного вида: берём правильное арифметическое выражение, написанное на Go, например, и вычёркиваем в нём все символы, кроме скобок. Вопрос – а мог ли получиться данный ряд скобок? Не случилось ли по дороге, что мы что-то напутали и получили невозможную последовательность скобок.

Скорее всего дети придумают способ с подсчётом: идём вдоль ряда, левая скобка - +1, правая - -1; во время прохода получаемая сумма должна быть всё время неотрицательной (>=0), а в конце сумма должна быть равна 0. И тут всё просто и естественно, даже не будем выжимать детей на обоснование такого решения - крайне маловерятно, что они выдадут что-то осознанное, а, самое главное, дальше всё станет ясным из более общего случая.

Более общий случай рассмотрим такой: а что будет, если имеется несколько сортов скобок? Тогда способ с подсчётами не прокатывает, например: ([)]. Дадим немного времени детям попридумывать какие-нибудь методы решения, построим контрпримеры к их «алгоритмам», в общем, поиграемся немного в эту игру. Чтобы дети разогрелись и осознали проблему, а то с одним видом скобок действительно работает решение, получаемое почти бессознательно.

Обсуждение (алгоритма) решения

А потом начнём думать. А что же такое скобки вообще, что они и для чего? А скобки нужны для того, чтобы определять порядок выполнения операций. Как

мы вычисляем значение арифметического выражения? А вот как: берём любую пару однотипных скобок - левую и правую, между которыми других скобок нет, вычисляем значение выражения в этих скобках, и заменяем всё выражение на полученное значение, а пару скобок вычёркиваем. Повторяем процесс, пока не останется одно число (вообще говоря, не обязательно число, но это совсем неважно сейчас). Ага! Вычёркиваем пару соответствующих скобок - однотипную пару левая скобка-правая скобка, между которыми нет других скобок. Интересно. В конце концов, понятно, скобок остаться не должно. Вот вам и алгоритм: находим пару соответствующих скобок (`(` и `)`, или `[` и `]`, или `{` и `}`, да каких угодно) и вычёркиваем её. Если вычеркнуть ничего не удаётся, а скобки ещё остались, то значит скобочная строка была некорректной, если всё вычеркнули – то корректной.

Чудесный алгоритм: пробегаем вдоль строки скобок, ищем пару соотвествующих (напомню, что пара соответствующих скобок - это однотипная пара левая скобка-правая скобка, между которыми нет других скобок) и вычёркиваем её.
Если от строки ничего не осталось, то строка была корректной, если что-то осталось, а вычеркнуть ничего не удаётся - то некорректной. Легко и просто. Напрягает немного то, что после вычеркивания надо бы начинать поиск соответствующей пары с самого начала. Нет, я понимаю, что можно поизвращаться и придумать какую-нибудь эвристику, чтобы не возвращаться к самому началу, а делать только шажок-другой назад, но давайте этот этап проскочим. Пусть даже и с лёгким насилием над детьми - ещё чуть-чуть, и всё встанет на свои места.

Построение алгоритма

Давайте посмотрим чуть дальше... Вот мы читаем строку скобок слева направо. Вот пришла какая-то правая скобка. Перед ней стоит некая левая скобка (почему не правая? сейчас увидите - правые скобки мы вовсе не будем хранить). Если они друг другу соответствуют, то можно их сразу вычеркнуть – пара соотвествующая, между ними уже никто влезать не станет. Если не соответствуют, то можно прекращать обработку – эту пару мы никогда не вычеркнем. Прекрасно. Итак, правая скобка – либо сразу вычёркивается вместе с соответствующей левой (которая в данный момент стоит на самом краю), либо вообще прекращается обработка с ответом “некорректно”/“fail”. Значит, мы будем хранить только левые скобки. Причём нам необходимо обеспечить такие способы доступа:

1. добавить левую скобку в конец строки - *Push*
2. посмотреть, какая скобка стоит в конце строки - *Get*
3. выдернуть последнюю скобку в строке (именно выдернуть - забрать последний элемент строки - из строки удалить, и нам выдать) - *Pop*
4. и ещё одно действие - проверить, есть ли скобки в строке, не пустая ли она. Эта проверка обязательно нужна, а то придёт правая скобка, а мы полезем в пустой ряд за левой скобкой. Неловко получится - *Empty*.

Такая структура данных (АТД - абстрактный тип данных) называется стек (stack) или LIFO (Last In – First Out).

Обсуждение реализации - 1. Стек на слайсе.

Понятно, как реализовать стек с помощью слайса. Тут даже и говорить практически не о чем - всё очевидно. Так что и не будем говорить, а просто посмотрим на программу. Читает она данные из файла `brackets.dat`, который содержит скобочные выражения для проверки - по одному выражению в строке. **Программа brackets_01a.go.**

```
package main

import     (
    "fmt"
    "errors"
)

func Pop (pstack *[]rune) (rune, error)   {
    if Empty(*pstack)  {
        return 0, errors.New("")
    }  else  {
        c:= (*pstack)[len(*pstack)-1]
        *pstack = (*pstack)[:len(*pstack)-1]
        return c, nil
    }
}

func Push (pstack *[]rune, r rune) {
    *pstack = append(*pstack, r)
```

```
}

func Empty(stack []rune) bool  {
    return len(stack) == 0
}

func check(brackets string) bool  {
    var stack []rune
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            Push(&stack, b)
        case ')':
            if top, err:= Pop(&stack); err != nil || top != '('  { return false
}
        case ']':
            if top, err:= Pop(&stack); err != nil || top != '['  { return false
}
        case '}':
            if top, err:= Pop(&stack); err != nil || top != '{'  { return false
}
        default:
            return false
        }
    }
    return Empty(stack)
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
```

```
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

И сразу же перепишем эту программу в таком, более энергичном стиле - заведём тип stack (который есть слайс скобок) и инкапсулируем в него соответсвующие методы: Pop, Push и Empty. Get оказывается не у дел, по уму надо бы его написать, но, во-первых, это тривиально, а во-вторых, этот текст - это пример, так что не будем рассеивать внимание читателей.
Программа brackets_02a.go.

```
package main

import     (
    "fmt"
    "errors"
    "os"
)

type stack []rune

func (s *stack) Pop () (rune, error)   {
    if s.Empty()  {
        return 0, errors.New("")
    }  else  {
        c:= (*s)[len(*s)-1]
        *s = (*s)[:len(*s)-1]
        return c, nil
    }
}

func (s *stack) Push (r rune) {
    *s = append(*s, r)
}
```

```
func (s stack) Empty() bool  {
    return len(s) == 0
}

func check(brackets string) bool  {
    var s stack
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            s.Push(b)
        case ')':
            if top, err:= s.Pop(); err != nil || top != '('  { return false }
        case ']':
            if top, err:= s.Pop(); err != nil || top != '['  { return false }
        case '}':
            if top, err:= s.Pop(); err != nil || top != '{'  { return false }
        default:
            return false
        }
    }
    return s.Empty()
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
```

```
}
```

Здесь есть один очень важный **технический момент**, про который обязательно надо поговорить хоть чуть-чуть -

Передача ресивера по адресу vs передача ресивера по значению

На прошлом занятии был повод немножко поговорить про передачу по адресу. И главное мы поняли - если мы хотим, чтобы изменения формального параметра (того,который живёт в функции) не отражались на фактическом параметре (того, коорый живёт в вызывалке), то передаём параметр по значению, если хотим, чтобы отражались - передаём по адресу. С ресиверами методов всё обстоит абсолютно точно также. В примере `brackets_01b.go` методы Push и Pop совершенно естественно получают ресивер по адресу, а метод Empty - по значению (кстати, и метод Get, если бы мы его написали, тоже). Но у методов есть одно существенное отличие:

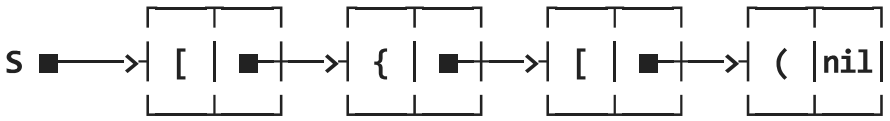
> **Когда функция получает аргумент-значение, она получает только значение фактического параметра.**
> **Когда функция получает аргумент-адрес, она получает только адрес фактического параметра.**
>
> **Когда метод получает ресивер - значение или адрес, - он получает и значение фактического ресивера, и его адрес.**

На реализацию методов это никак не влияет, а вот вызов методов становится попроще, и мы это хорошо видим на данной паре примеров. Живёт в функции Check стек s, ну, и чудесно - он выполняет свои методы, а мы можем даже не задумываться, адрес нужно передавать или значение.

Обсуждение реализации - 2. Приходим к идее связных структур.

А давайте теперь ещё подумаем. Что нам надо от наших хранимых скобок?

Добавлять скобку в конец и смотреть на последнюю скобку. Ок, это легко. А ещё надо выдёргивать последнюю скобку. Выдернуть-то мы её выдернем, но дальше-то жизнь продолжается. И нам после этого понадобится то ли смотреть на новую последнюю (бывшую предпоследнюю) скобку, то ли выдёргивать её, то ли к ней прицеплять новую скобку. Всё так. Но! Ничего больше, кроме того, что после выдёргивания последней скобки нужно иметь доступ к новой последней скобке. В самом деле, у меня лично использование слайса для хранения стека вызывает ощущение избыточности - нам совсем не нужно в любой момент времени иметь прямой непосредственный доступ к любой хранимой скобке. Повторю, всё что нам надо - это после выдёргивания последней скобки иметь доступ к новой последней скобке. Т.е. последняя скобка должна давать нам доступ к предпоследней. Каждая скобка должна давать доступ к своей предшественнице. И ничего больше. Так пусть каждая скобка просто смотрит на предыдущую. Смотрит - это значит хранит где-то у себя её адрес. Бинго! Давайте к каждой скобке прицепим адрес её предшественницы, указатель на неё. Этому нехитрому требованию полностью удовлетворяет вот такая структура:



Начальная скобка в стеке не смотрит ни на что - за ней ничего нет, так что вполне естественно сказать, что она смотрит на nil. На последнюю скобку смотрит переменная S, собственно мы всю структуру и держим через переменную S. Да, а стек на картинке растёт справа налево - так получилось...

Всё получилось - всё срослось просто идеально. Мы в любой момент времени видим непосредственно только последнюю скобку; выдёргивая последнюю скобку, мы можем зацепиться за предпоследнюю, она же новая последняя; добавить новую скобку к текущей последней - тоже не проблема.

Да, мы пришли к идее связной структуры.

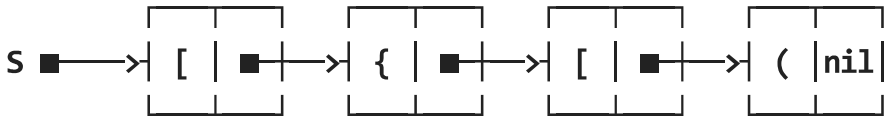
Пара общих слов о реализации связных структур

Итак, идея связных структур: хранить данные связанно, т.е. вместе с данными хранить какие-то связи между ними. Разумеется, связи - это адреса, они же указатели. Как именно организовывать связи зависит от тех методов доступа,

которые должна нам обеспечить структура, от предназначения структуры. Как придумывать связные структуры? Будем учиться. Думать, иначе говоря. Будем рассматривать разные задачи и строить какие-то структуры данных, более или менее адекватные задаче и алгоритму её решения. Связные структуры позволяют конструировать очень гибкие конструкции.

А сейчас заметим, что вместе с данными мы храним какие-то связи между ними. Это значит, что каждый узел нашей структуры, каждая единица, каждый элемент, называйте как хотите, должен состоять из разнотипных величин - из данных и из указателей. Но всё-таки хотелось бы сохранить их единство - всё-таки речь идёт об элементе структуры. Выход очевидный - используем для представления узла тип struct, который и позволяет удобно и естественно хранить совместно разнотипные данные.

Следующая программа (`brackets_01b.go`) в точности реализует описанный подход и в точности соотвествует вышеприведённому рисунку. Повторю его здесь:



Конечно, надо очень тщательно, рисуя картинки на доске, разобрать в динамике функции Push и Pop. С функцией Empty всё проще, но и про неё что-то сказать необходимо. Тем более, что пустой стек - это очень важный пограничный случай, кот орый должен обрабатываться особо, по крайней мере при выталкивании верхушки стека (Pop). Не буду здесь расписывать объяснения - они довольно очевидны, но этот момент крайне важен, ему надо уделить максимум внимания, это принципиальнейший момент, хоть это и чистая техника. Но! Очень важно рисовать текущую/желательную ситуацию, а ещё важнее, чтобы дети рисовали. ЭТО ОЧЕНЬ ВАЖНО.

Программа brackets_01b.go.

```
package main

import     (
```

```
    "fmt"
    "errors"
    "os"
)

type  (
    node struct {
        bracket rune
        next *node
    }
    stack *node
)

func Pop (ps *stack) (rune, error)   {
    if Empty(*ps)  {
        return 0, errors.New("")
    }  else  {
        c:= (*(*ps)).bracket
        *ps = (*(*ps)).next
        return c, nil
    }
}

func Push (ps *stack, r rune) {
    p:= new(node)
    (*p).bracket = r
    (*p).next = *ps
    *ps = p
}

func Empty(s stack) bool  {
    return s == nil
}

func check(brackets string) bool  {
    var s stack
```

```
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            Push(&s, b)
        case ')':
            if top, err:= Pop(&s); err != nil || top != '('  { return false }
        case ']':
            if top, err:= Pop(&s); err != nil || top != '['  { return false }
        case '}':
            if top, err:= Pop(&s); err != nil || top != '{'  { return false }
        default:
            return false
        }
    }
    return Empty(s)
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

И опять переписываем нашу программу так, чтобы функции доступа к стеку были реализованы как методы типа stack. Здесь есть один немножко странноватый технический момент: мы не можем объявит еип stack просто указателем на верхушку стека, как это сделано в предыдуще программе. Это потому, что указатиельный тип не может иметь методов. Просто синтаксически это

запрещено. Думаю, что это связано с уже упоминавшимся дуализмом параметра-ресивера - он и значение, он и адрес, но это мои предположения. Во всяком случае такое есть, но бороться с этим совсем легко - вместо адресного/пойнтерного/указательного типа заводим struct с единственным полем этого адресного типа. Как это сделано - в **программе** brackets_02b.go.

```
package main

import  (
    "fmt"
    "errors"
    "os"
)

type lmnt struct {
    bracket rune
    next *lmnt
}

type stack struct {
    head *lmnt
}

func (s *stack) Push(bracket rune) {
    s.head = &lmnt{bracket, s.head}
}

func (s *stack) Pop() (rune, error) {
    if s.head == nil {
        return 0, errors.New("List is empty")
    }
    c:= s.head.bracket
    s.head = s.head.next
    return c, nil
}
```

```
func (s stack) Empty() bool  {
    return s.head == nil
}

func initStack() stack {
    return stack{nil}
}

func check(brackets string) bool  {
    s := initStack()
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            s.Push(b)
        case ')':
            if top, err:= s.Pop(); err != nil || top != '('  { return false }
        case ']':
            if top, err:= s.Pop(); err != nil || top != '['  { return false }
        case '}':
            if top, err:= s.Pop(); err != nil || top != '{'  { return false }
        default:
            return false
        }
    }
    return s.Empty()
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
```

```
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

Принципиальных отличий от предыдущей программы здесь нет, main вовсе не отличается, но технический момент всё-таки несколько отличается, детям непросто разобраться со всеми этими звёздочками, а с этим надо не только разобраться, но и привыкнуть. А тут, как водится, способ один - упраженения. Ну, в данном случае, ещё и рисунки-рисунки-рисунки и ещё раз рисунки. Что есть, что надо, динамически перерисоовываем текущую ситуацию и т.д.

Задача о максимуме на плавающем окне - очередь.

Сразу отмечу, что здесь будет разобрано естественное, но довольно неэффективное решение. Зато оно позволяет нам рассмотреть очередь (queue). Эффективное решение, использующее дек, рассмотрим в следующий раз - в это раз точно это решение не вписывается, тем более, что оно менее очевидное, там ещё надо с алгоритмом разобраться. В общем всё срастается - сегодня очевидное решение и очередь, а менее очевидное решение, которое - очень удачно - использует родственную, но другую структуру - дек, пойдёт на следующий раз. Тем более, что очередь реализуется на базе односвязного списка, который у нас уже есть, а дек требует уже двусвязного списка, и это тоже удачно получается.

Описание задачи

А задача формулируется совсем просто: есть ряд из N чисел, и есть окно на K элементов, K < N. Окно двигается слева направо от начала ряда чисел к концу, и каждый раз нас интересует максимальное значение, видное в данный момент в окне. Понятно, что на выходе мы ожидаем N-K+1 максимумов. Например, если мы имеем ряд из 7 чисел

```
4 7 5 3 6 11 9,
```

а K = 3, то на выходе мы получаем

```
7 7 6 11 11
```

Обсуждение решения (алгоритма) и построение алгоритма

Ну, что, алгоритм мы здесь применим тривиально простой: просто двигаем окно и каждый раз вычисляем максимум в этолм окне. Посмотрим, как изменяется содержимое окна. Первые K шагов окно наезжает на начало ряда чисел и увеличивается в размерах. При этом максимум вычислять пока не надо (кроме последнего раза, но перенесём это действие на следующий этап). Вот что происходит:

```
1-й шаг. Окно содержит: 4
2-й шаг. Окно содержит: 4 7
3-й шаг. Окно содержит: 4 7 5
```

И теперь начинается второй этап - движение заполненнного окна. Первым делом вычисляем максимум в полученном окне, получаем 7, а затем продвигаем окно. При том в конец добавляется следующее число из данного ряда чисел - это 3, а начальное число - это 4 - уходит из окна. Получаем в итоге содержимое окна: 7 5 3. И т.д, пока окно не дойдет да конца ряда чисел: 5 3 6, 3 6 11 и 6 11 9.

Какие действия в итоге мы производим:

```
- добавляем число в конец окна
- убираем число из начала окна
- вычисляем максимум чисел в окне
```

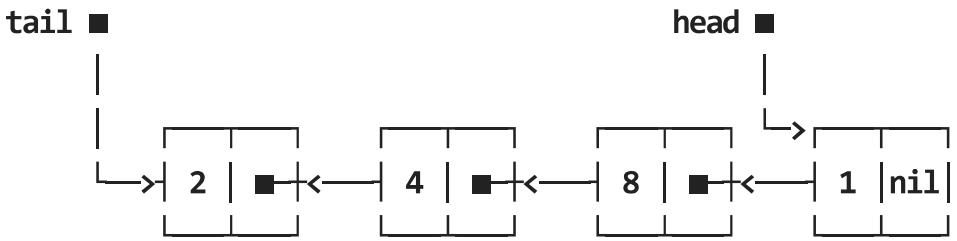
Всё. Такая структура данных (без поиска максимума) называется, естественно, очередью (queue) или FIFO (First In - First Out).

Обсуждение реализации

Конечно, очередь прекрасно реализуется с помощью слайса. Хотя тут уже

появляется некоторый тормозящий момент - необходимость удалять элемент из начала слайса, а не с конца, а это уже хммм... Так что есть смысл написать два решения, побенчмарчить и сравнить их.

Итак, посмотрим на реализацию очереди, думая о связном списке.
Ну, максимум вычислять будем по-простому - пробегать по списку чисел в окне и выбирать наибольшее, тут вопросов нет, неважно, слайс у нас или список. Как добавлять число в конец списка чисел в окне тоже понятно - берём последнее число и пристраиваем к нему новое число. А вот убирая первое число в списке, мы должны определить новое первое число - бывшее второе. А это значит, что первое число должно быть связано со вторым. А, поскольку каждое число в конце концов станет первым (ну, почти каждое, последние числа не в счёт) , то каждое число долюно хранить у себя связь со следующим числом. Отлично. Работает тот же самый связный список. Одно отличие, тем не менее, есть - нам нужно видеть не только первое число в списке, но и последнее. Да, конечно мы можем пробегать по всему списку в поисках последнено числа, но как-то это уж совсем странно выглядит, лучше уж мы будем держать наш список чисел в окне за два места - за начальное число и за последнее. Как-то так:



Ага, хранение очереди подразумевает, в отличие от хранения стека, держать два указателя - на голову списка (head) и на хвост (tail). Вот и прекрасно. Очередь - это struct из двух указателей.

Приведу обе реализации: одна реализует очереди на слайсе, вторая - на связном списке. В обеих программах обработку очереди реализуем в виде методов типа queue, реализовать способы обработки с помощью просто функций не будем, ну их. Вот эти реализации
numbers_01.go. Реализация очереди слайсом.

```
package main
```

```
import     (
    "fmt"
    "errors"
    "os"
    "math"
)

type queue []int

func initQueue() queue  {
    return make([]int, 0, 0)
}

func (q *queue) Add (n int) {
    *q = append(*q, n)
}

func (q queue) Max () int {
    res := math.MinInt64
    for _, x := range q  {
        if x > res  { res = x }
    }
    return res
}



func (q *queue) Remove () error   {
    if (*q).Empty()  {
        return errors.New("Attempt to remove from empty queue")
    }  else  {
        *q = (*q)[1:]
        return nil
    }
}

func (q queue) Empty() bool  {
```

```
    return len(q) == 0
}

func main() {
    f, err := os.Open("numbers.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    q:= initQueue()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }
        q.Add(x)
    }
    for {
        fmt.Println(q.Max())
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }
        q.Add(x)
        q.Remove()
    }
}
```

numbers_02.go. Реализация очереди связным списком.

```
package main

import     (
    "fmt"
    "errors"
    "os"
```

```
    "math"
)

type lmnt struct {
    n int
    next *lmnt
}

type queue struct {
    head *lmnt
    tail *lmnt
}

func initQueue() queue  {
    return queue{nil, nil}
}

func (q *queue) Add (n int) {
    if (*q).Empty()  {
        (*q).tail = &lmnt{n, nil}
        (*q).head = (*q).tail
    } else {
        (*(*q).tail).next = &lmnt{n, nil}
        (*q).tail = (*(*q).tail).next
    }
}

func (q *queue) Remove () error   {
    if (*q).Empty()  {
        return errors.New("Attempt to remove from empty queue")
    }  else  {
        (*q).head = (*(*q).head).next
        return nil
    }
}
```

```
func (q queue) Max () int {
    res := math.MinInt64
    runner:= q.head
    for runner != nil  {
        if (*runner).n > res  { res = (*runner).n }
        runner = (*runner).next
    }
    return res
}

func (q queue) Empty() bool  {
    return q.head == nil
}

func main() {
    f, err := os.Open("numbers.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    q:= initQueue()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }
        q.Add(x)
    }
    for {
        fmt.Println(q.Max())
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }
        q.Add(x)
        q.Remove()
```

```
    }
}
```

Общая технологическая линия решения задач кратко.

1. Нашли решение задачи – выяснили, какие операции с данными нам нужны. Набор операций, набор способоы доступа к данным - это, фактически, и есть описание структуры данных.
2. Придумали по возможности наиболее адекватный способ связывания/хранения данных, порисовали картинки.
3. Технические вопросы в это раз очень важны: рисуем, причём двумя, а то и тремя цветами, причём всё время в динамике рисуем, постоянно видим, что у нас есть, что будет, что надо сделать. Настоятельно требуйте, чтобы дети на практике тоже рисовали, чтобы не держали картинку в голове – обязательно запутаются на пустом месте. Рисунок, причём динамический, позволяет избежать массы ошибок и трудностей.
4. Особый адрес – nil. Его применяют обычно как заглушка, признак окончания движения по связной структуре данных, признак терминального элемента. Бывают и другие варианты применения nil, но довольно редко.

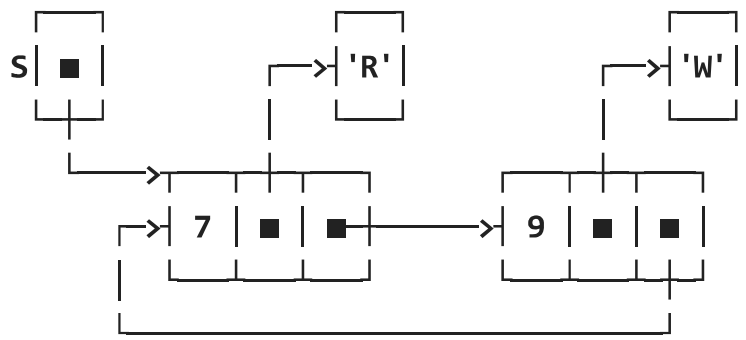
Практика.

Упражнения “создать конструкцию”

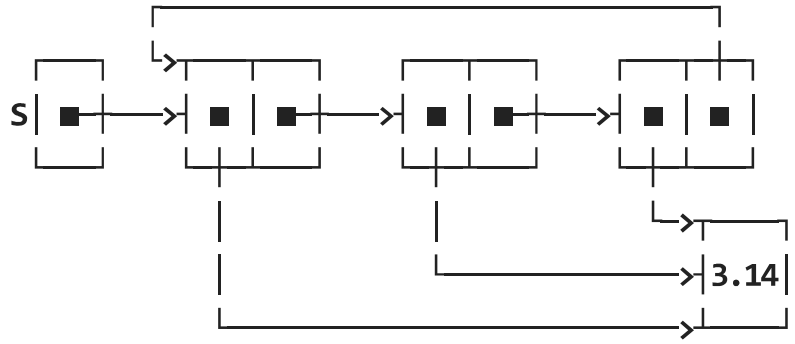
А на практику хорошее упражнение - дети его обычно делают в конце концов, хотя и помучаться приходится. Но многое им становится понятно, упражнение полезное. Можно все не давать - вполне достаточно двух-трёх, можно какие-то упростить или изменить, но идея понятна. Беда здесь в том, что проверить правильность кода можно только анализируя код - это требует сильной активности от преподавателя. Но упражнение очень... Хотя геморроя много и детям, и преподам.
И упражнение такое. Создать конструкции, изображённые на рисунках, объявив для этого соответствующие типы, столько, сколько надо. А вот переменную

можно использовать только одну - переменную S. Да, и эта S есть переменная типа struct с одним полем, напоминаю. Можно для простоты, хоть это и несколько неуклюже, считать, что все прямоугольники на рисунках - есть struct'ы, и не только состоящие из 2-3 полей, но и из одного. Хотя это и не обязательно.

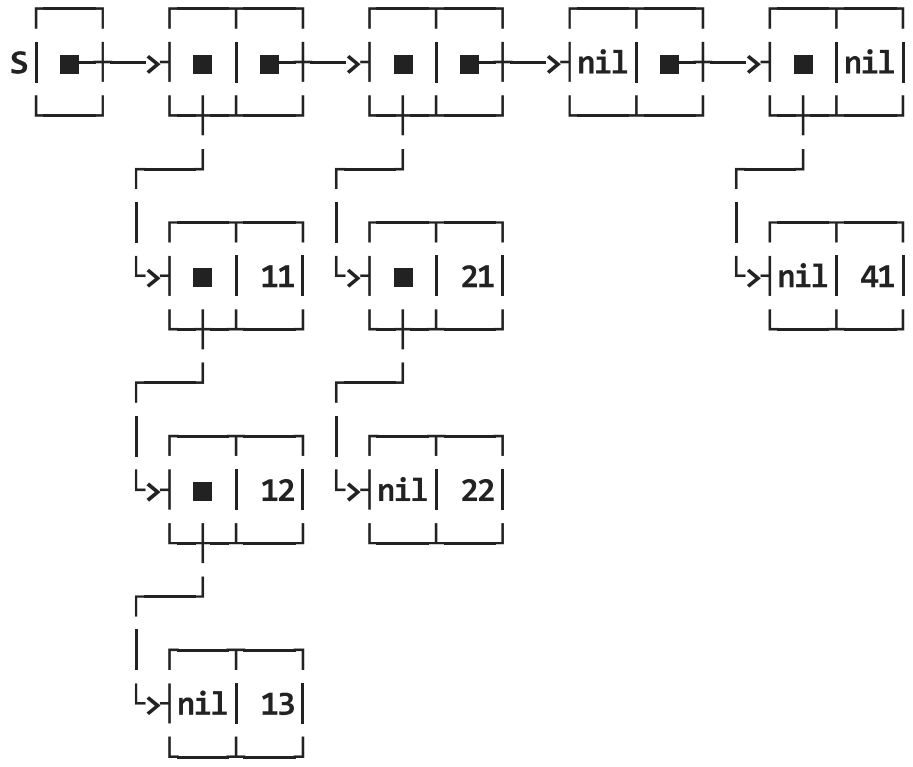
1.



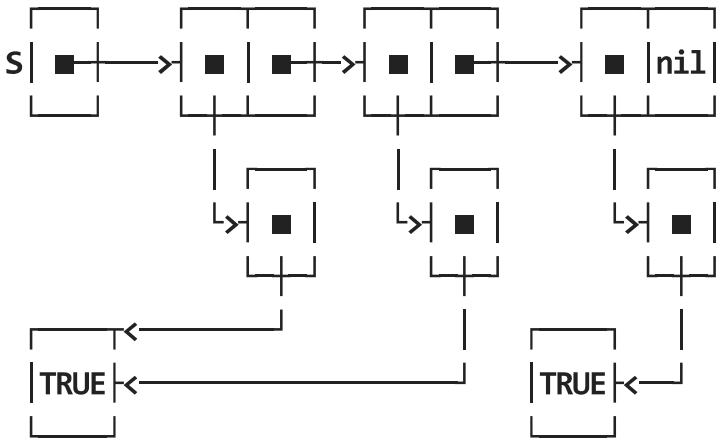
2.



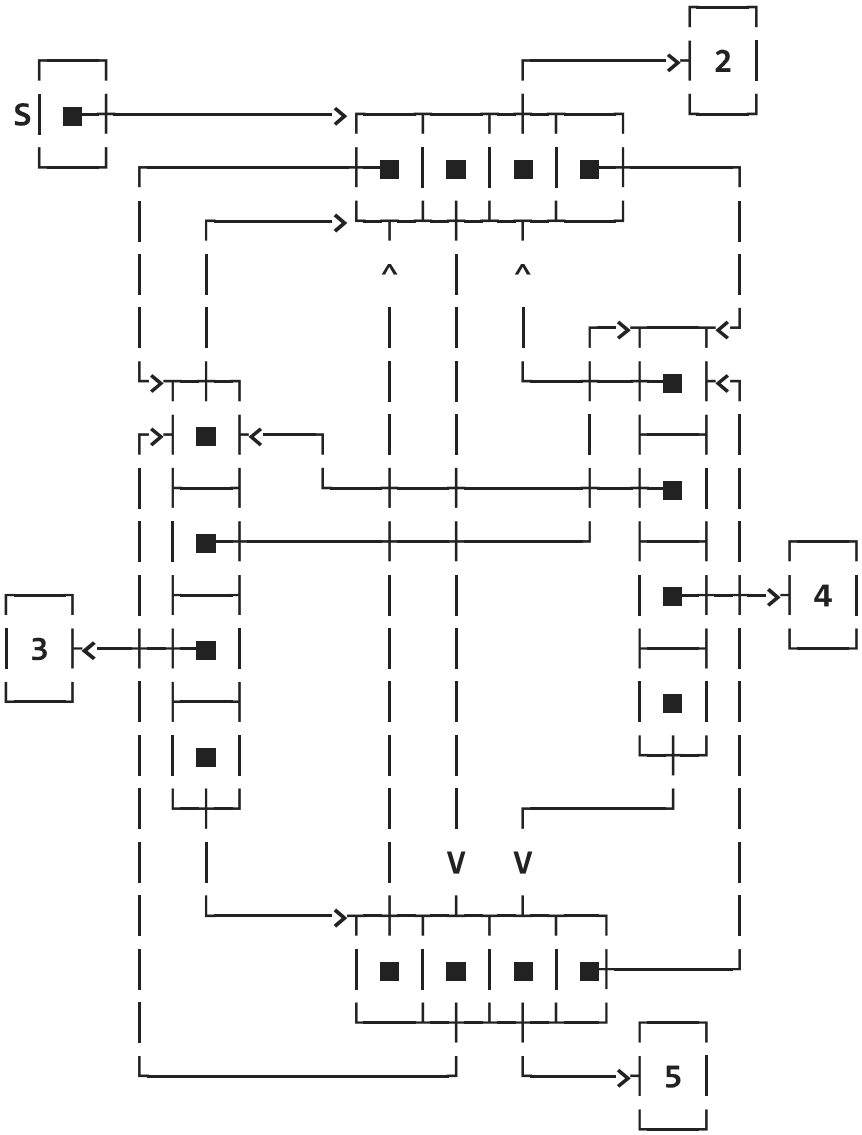
3.



4.

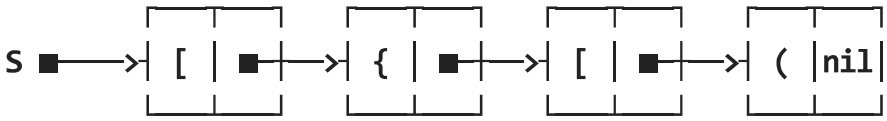


5. Этот рисунок тоже сюда вставлю, раз уж он есть, но это уже немного зашквар, хотя сделать его интересно.



Задачи на работу со связным списком - linkedList

И ещё, тоже очень хорошая серия, к тому же она нам пригодится в следующий раз - куча задач на реализацию всяких функций над связным списком. При этом связный список понимаем в том смысле, в каком он использовался в задаче о скобках: список имеет непосредственный доступ к последнему элементу списка. Последнем - имеется в виду не последнему вставленному, последнему не хронологически, а географичеки, на которого никто не смотрит из элементов списка, на который смотрит переменная S на нашем рисунке. Повторю рисунок:



Я говорю о последнем географически потому, что в связный список мы можем вставлять элементы в середину или даже в конец списка, только до соответсвующего эместа сначала надо добежать, начиная с последнего географически. И удалять тоже можно элементы из середины списка. Иначе говоря, объявляем типы

```
type  (
    lmnt struct {
        n int
        next *lmnt
    }

    linkedList struct {
        head *lmnt
    }
)
```

и пишем функции. Я привожу здесь заголовки и краткие спецификации. Да, и все функции реализуем как методы типа linkedList.

`func (l linkedList) Len () int`
Количество элементов в списке.

`func (l linkedList) Max () int`

Максимум списка.

`func (l linkedList) Min () int`

Минимум списка.

`func (l linkedList) CompareList ( list linkedList) bool`

Совпадает ли список list с ресивером l.

`func (l linkedList) InsertKth (x int, k int)`

Вставить число x на k-ю позицию, k > 0. Считаем, что элементы в списке пронумерованы с головы списка, и нумерация начинается с 0. Можно предусмотреть, что функция будет возвращать ошибку, если k слишком большое - больше длины списка. Либо возвращать bool - удалось или не удалось вставить.

`func (l linkedList) GetKth (k int) int`

Возвращает k-й элемент списка, k >= 0. Как и в предыдущей функции, считаем, что элементы в списке пронумерованы с головы списка, и нумерация начинается с 0. Можно предусмотреть, что функция будет возвращать ошибку, если k слишком большое - больше длины списка. Либо возвращать bool - удалось или не удалось вставить.

`func (l linkedList) RemoveKth (k int) bool`

Удаляет из списка k-й элемент, k >= 0. И здесь, считаем, что элементы в списке пронумерованы с головы списка, и нумерация начинается с 0. Но тут уж точно функция должна возвращать, был в списке элемент с таким номером и его удалось убрать из списка, или нет.

Я не выписал тут методов, связанных с начальным элементом списка (Insert, Get, Remove), поскольку они уже у нас написаны. Но можно дать и их.

Ну, и т.д. Список таких задач можно продолжать очень долго.